On the Detection of Common Schemas in Programs

Richard Lee Denney

A-91

Consider listening to a computer program. The statements are read to you a line at a time and you are to try and understand the code. A difficult task indeed. Yet we understand problems of a different type everyday in the form of spoken English.
The problem of understanding spoken computer language is one of memory; our inability to remember more than a few lines of code at a time. Gruneberg and Morris [1979] say concerning reading and listening,

> "The integration of word's meanings from sentences is a task requiring storage of earlier presented words until later words are available..."

The problem is not unique to programming language understanding; an example of well known algorithms made complex by memory constraints is mental addition. Hitch [1978] performed experiments exploring the role of information storage in people's working memory in mental arithmetic and stated the probability of getting a correct answer in terms of memory decay and load.

So why are we able to communicate verbally with English? The following is from Jenson and Tonies [1979] (page 266), the subject is the use of pseudocode as a design langueage.

> Temperature data in degrees Kelvin is recorded in a file automatically at the rate of one measurement per minute. A subroutine is to be written to provide the average recorded terperature value over a one-hour period. If a zero reading exists in the data file, the mean value for that one-hour period is to be set to 0.
>
> A pseudocode solution of the function to be performed by the subroutine is

```
sum=0
minute=1
while (minute LTE 60)
        if (file(minute) NEQ 0)
                sum=sum+file(minute)
        else
                sum=0
                escape while
        end if
        minute=minute+1
end while
mean=sum/60
```

(end of Jenson and Tonies excerpt)

This same problem stated in English might read,
Sum the sixty elements of array File.
If any single element is zero, the sum is to be zero.
The average is the sum divided by 60, the number of minutes in an hour.

I tried reading the problem stated both ways to employees where I work and to no one's suprise, the English was easier to understand.
The consensus was that the single word SUM in the English version made the difference. Because everyone knew what it is to SUM, that a-priori knowledge could be used to shorten the description of what was to be done to FILE.

Schank and Abelson [1975] argue that people comprehend linguistic messages through the use of what is called in Artificial Intelligence and Psychology literature, schemas. A schema serves to summarize the typical phenomana which occur during an event, or as in our example above, summarize the computations in a process.
A restaurant schema is given by Schank and Abelson:

```
schema: Restaurant.
characters: Customers, hostess, waiter, chef, cashier.
Scene 1: Entering.
        Customer goes into restaurant.
        Customer finds a place to sit.
                He may find it himself.
                He may be seated by a hostess.
                        He asks the hostess for a table.
                        She gives him permission to go to the table.
        Customer goes and sits at the table.
Scene 2: Ordering.
        Customer receives a menu.
        Customer reads it.
        Customer decides what to order.
        Waiter takes the order.
                Waiter sees the customer.
                Waiter goes to the customer.
                Customer orders what he wants.
        Chef cooks the meal.
Scene 3: Eating.
        After some time the waiter brings the meal from the chef.
        Customer eats the meal.
Scene 4: Exiting.
        Customer asks the waiter for the check.
        Waiter gives the check to the customer.
        Customer leaves a tip.
                The size of the tip depends on the
                goodness of the service.
        Customer pays the cashier.
        Customer leaves the restaurant.
```

If a person is familiar with the typical restaurant schema, heuristic information is available to allow inference that is needed to understand what is being said, but which may not be explicitly stated in the verbal transmission.
For example upon hearing, "John didn't leave any tip", the listener can understand and reply with, "Oh, was the service that bad?"

The same view is expressed by Kintsch and Dijk, as paraphrased here from Atwood and Ramsey [1978], (pages 7-8)

> "... a schema functions like an outline with empty slots. That is, a schema is a set of expectations about what the story will consist of."

Scientists of Artificial Intelligence have tried to exploit commonalities of story schemas to enable computers to understand them.
Winston [1977], discusses the use of two such prototypes, News Articles and children's stories.
Other prototypal story schemas might be:

Mystery Story schema - crime committed, suspects introduced, clues given, and everyone knows the butler did it.
Fairy Tale schema - Once upon a time..., the story, and everyone lived happily ever after.
Algebra Problem schema - always begin with a bunch of facts. You know you'd better pay attention to them because algebra problems, unlike mystery stories, never give irrelevant information.
Situation Comedy schema - anyone familiar with TV is all too aware of the similar pattern all these programs follow.

The discussion of schemas and prototypes of schemas thus far has been to set a precedent for why it is useful to discover the same in computer programming languages.
Atwood and Ramsey [1978] (page 23-24),

> "The existence of macrostructures in text memory is easily demonstrated.... We have, for example, one macrostructure for journal articles, one for fairy tales, one for suspense stories, etc. With text, we detemine the appropriate macrostructure fairly quicly and easily and proceed to build upon it. When we consider programs, however, it is not so clear what types of macrostructures are involved... the lack of such prototypic structures could explain why learning to write and comprehend programming languages is often considered more difficult than learning to write and comprehend natural languages for which well defined prototypes exist... the discovery of a class of program macrostructures could greatly increase our knowledge of programming behavior."

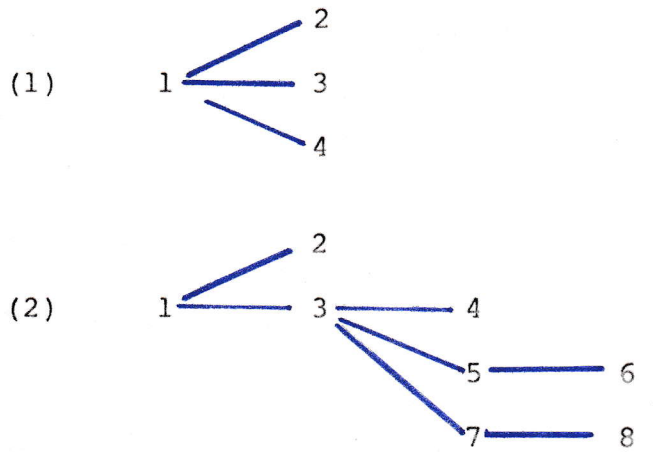
Atwood and Ramsey suggest the possibility of Logical Structures as a basis for detecting prototypes of schemas in programs. Their work was based largely on a study by Kintsch and Keenan [1973]. Concerning Logical Structures Kintsch and Keenan say, (page 257-258)

> "... a part of a sentence or paragraph must be identified as the theme; there are grammatical relations among the elements of a sentence; and finally, each text has logical structure... A number of theories have recently been developed concerning the logical structure of text and, closely related, the nature of semantic memory."

The means proposed to represent these Logical Structures is Propositional Hierarchies. I will refer back to the Atwood and Ramsey study for a description of Propositional Hierarchies.

> "The text base consists of a connected, partially ordered list of propositions. Propositions contain one or more arguments plus one relational term. Consider first the concept of a proposition. Basically a proposition makes a meaningful statement. It is convenient to think of a proposition as the last amount of textual material that can convey an idea... A text base consists of a partially ordered hierarcy of propositions. That is, one or more propositions will represent the superordinate, or most important, ideas to be expressed and other propositions{subordinate} will give more information about these ideas.
>
> To illustrate these concepts, consider the following two sentences (from Kintsch [1974]).
>
> (1) Romulus, the legendary founder of Rome, took the women of the Sabin by force.
>
> (2) Cleopatra's downfall lay in her foolish trust in the fickle political figures of the Roman world.
>
> The suggested propositions underlying the first sentence are:
>
> 1. (took, romulus, women, by force)
>
> 2. (found, romulus, rome)
>
> 3. (legendary, romulus)
>
> 4. (sabine, women) ...

The hierarchies of these sentences are



(End of Atwood and Ramsey excerpt. Above in {} is mine)

The thrust of the Atwood and Ramsey report is to apply this dividing into Propositional Hierarchies to programs to detemine if the effort involved in finding bugs is affected by the depth at which the bugs occur in the hierarchy. FORTRAN programs studied had to first be converted into propositional form using a convention they set forth.

The balance of this paper is a case study to suggest the possibility of the use of propositional hierarchies as a criterion for defining program features, and programming styles, which in turn may be useful in defining common program schemas.

Assembly language, DEC-10's MACRO-10, is used to eliminate the need to reduce the code to propositions. Assembly language is already proposition like; each statement typically consists of two or more arguments and the relation between them. For example,

```
CAME T1,MEMLOC          ;Skip if contents of
                        ;T1 is equal to that of MEMLOC.
                        ;Arguments are T1 and MEMLOC.
                        ;Relation is "skip if equal".
```

Propositional Hierarchies are plotted using SPSS scattergrams. The Y and X axis are graduated equally from zero to the number of propositions (program statements) in the program.
The Y-axis is for Superordinate Proposition number and the X-axis for the Subordinate Proposition number.
Please note that the resolution of the scattergrams is not good enough to allow each graduated mark on the X or Y axis to correspond to a single proposition in the program. This is evidenced by the fact that the numbers(instead of stars) on the scattergram indicate a number of stars plotted at a single coordinate point.

Disregarding this fact, a stars coordinates (y,x) can be interpreted as proposition y is the Superordinate to x, or conversely proposition x is the Subordinate to y.
Using program listings it is in fact possible to determine the exact propositions that most stars represent.

LOCALIZATION

The most prominent feature of these scattergrams is the diagnol running from bottom left to upper right. Reference scattergram for program D2D.
Because the X and Y axis are graduated equally, a diagnol with slope 1 would be stars with coordinates (1,1), (2,2), (3,3), etc.
These points themselves are not plotted since it would require statements being subordinate to themselves. However, stars that fall very close to this diagnol represent propositions that are close to their superordinate proposition.

Consider this code example,

```
movei a,4
multi a,b
add   b,c
div   c,d
```

Each proposition is subordinate to the proposition immediately above it; let me call this trait Localization. It is the degree to which thoughts created by a superordinate proposition are used by the subordinates in the same local of the program. This is an objective of modularization; confining related thoughts to a single understandable module.

The slope figures(computed by SPSS) at the bottom of the scattergrams are a measure of the programs overall localization. As the localization improves, the slope will approach 1.
As a complexity measure a slope of one should mean the code is very easily understood, e.g.

```
setz    a,      ;The localization
setz    b,      ;of this code is
movei   c,-1    ;indicated by the
seto    d,      ;slope which is 1.
```

The propositions in this example are subordinate to no other propositions.

SPOTTING LOCALIZED CODE

Looking at the scattergram for program TPSLEW we are able to detect a section of code that has ideal localization. Note the stars circles in the lower right. Looking at this pattern we might guess that the proposiitional structure is such that subordinate propositions occur very soon after their superordinates (they fall close to the diagnol) and that they occur no where else in the program (no stars to the right); ideal localization as demonstrated in a previous example. Looking at the code that produced this patter we see that this is indeed the case:

```
02800   BUF1:   XWD     ^D1100,BUF2
02900           BLOCK   ^D1100+2
03000           Z
03100   BUF2:   XWD     ^D1100,BUF3
03200           BLOCK   ^D1100+2
03300           Z
03400   BUF3:   XWD     ^D1100,BUF4
03500           BLOCK   ^D1100+2
03600           Z
03700   BUF4:   XWD     ^D1100,BUF5
03800           BLOCK   ^D1100+2
03900           Z
04000   BUF5:   XWD     ^D1100,BUF6
04100           BLOCK   ^D1100+2
04200           Z
04300   BUF6:   XWD     ^D1100,BUF7
04400           BLOCK   ^D1100+2
04500           Z
04600   BUF7:   XWD     ^D1100,BUF8
04700           BLOCK   ^D1100+2
04800           Z
04900   BUF8:   XWD     ^D1100,BUF9
05000           BLOCK   ^D1100+2
05100           Z
05200   BUF9:   XWD     ^D1100,BUF10
05300           BLOCK   ^D1100+2
05400           Z
```

BUF1 is superordinate to BUF2, which in turn is superordinate to BUF3 which is likewise to BUF4, and so on.

ACCUMULATOR USE CHARACTERISTICS

Across the bottom of each scattergram is a dense region of stars. This is produced by a style of programming forced on the coder by the language. It is caused by the repeated use of the accumulaters.
The necessity of using accumulators ties into the same propositional structure modules which should be logically unrelated.

Just as the mystery story schema suspects the butler, all assembly language hackers suspect bashed accumulators when modifications break formerly functioning code.

The horizontal bands of stars occur at the bottom because accumulator definitions typically occur early in the program.

TEMP STORAGE PROGRAMMING

Other horizontal bands may indicate the practice of overusing temporary storages. These occur higher on the Y-axis than the accumulator bands, and stretch from the diagnol to the right margin.
These are not temporary storages created for a specific task; these would tend to fall along the main diagnol since they would probably be local to their superordinate proposition. Rather, this is temporary storage which might have originally been created for a specific use, but was used again and again throughout the program rather than create other temporary storages when needed.

The yellow line on program GETDAY's scattergram is such a case. The storage is a variable called N; one of three the coder defined as M,N, and K and commented as "place to store ACs".
What happened to I,J, and L ?!

As with acccumulator use, this practice ties together in the propositional structure, code that should be kept logically seperated.

DECLARATION/COMPUTATION DIVISIONS

Reference program D2D's scattergram. The circled blank area in the lower left corner is caused by a particular style of programming; the division of declaration from computation.
The first 180 or so lines of the program are variable declarations and initializations. It is not untill after this point that subordinate propostions begin to occurr.
Program COMSTR exhibits this same trait.
The opposing style to this would be the definition and initialization of variables as needed.

LOGICAL SEPARATION OF PROCESSES

Reference the scattergram for INIT(the initilization part of D2D). Notice the circled yellow areas, isolated from each other and not overlapping on the X-axis. This type of pattern is caused by a program containing several propositional structures which are logically disjoint. This is in fact the case with INIT. It is composed of three separate initialization routines.
Note that since this routine was pulled out of D2D, the line numbers on the axis will not correspond with line numbers in the listing. Three routines mentioned here are GETDSK, SATGET, and SUFINI.

TPSLEW has partial seperation of processes as evidenced by the blank area extending from 80-160 on the Y-axis.

SUMMARY

Schemas appear to be a method by which people understand events. The development or discovery of common schemas in programming languages could increase our ability to understand programs. Given that propositional structures could be one criterion for defining common schemas, I have attempted to spot trends that occur in the propositional hierarchies of assembly language programs, using graphical representation.

/

Atwood and Ramsey, [1978], "Cognitive Structures in the Comprehension and Memory of Computer Programs: An Investigation of Computer Program Debugging" U.S. Army Research Institute Technical Report, tr-78-a21

Gruneberg and Morris, [1979], "Applied Problems in Memory", Academic Press

Hansen, W.J. [1978], "Measurement of Program Complexity by the Pair" SIGPLAN Notices, Vol 13 No. 3, March 1978, pg 61-64

Hitch, G.J., [1978], "The Role of Short-term Working Memory in Mental Arithmetic", Cognitive Psychology, October 1978, pg 302-323

Jenson and Tonies, [1979], "Software Engineering", Prentice-Hall

Kintsch and Kennan, [1973], "Reading Rate and Retention as a Function of the Number of Propostions in the Base Structure of Sentences" Cognitive Psychology, 1973, 5, pg 257-274

Kintsch, [1974], "The Representation of Meaning in Memory" Erblaum, 1974

Schank and Abelson, [1975], "Scripts, Plans, and Knowledge" Proceedings from Fourth International Conference on Artificial Intelligence

Weisberg, R.W. [1980], "Memory, Thought, and Behavior", Oxford Unv. Press

Winston, P.H. [1977], "Artificial Intelligence", Addison-Wesley Publishing

.